A Data Centered Approach for Cache Partitioning in Embedded Real-Time Database System

HU WEI, CHEN TIANZHOU, SHI QINGSONG, JIANG NING
College of Computer Science
Zhejiang University
College of Computer Science, Zhejiang University, Hangzhou, Zhejiang, 310027, P.R.China
P.R.China
ehu@zju.edu.cn http://embedded.zju.edu.cn

Abstract: - Embedded real-time databases become a basic part of the embedded systems in many using environments. Caches are used for reducing the gap between processor and off-chip memory. But caches introduce unpredictability in general real-time systems. Although several cache partitioning approaches have been purposed to tackle this problem, there is no scheme designed for real-time database system up to now. In this paper, we present a data centered cache partitioning approach that allows different tasks to have a shared locking partition in cache. The hard real-time tasks will have their own partitions and thus they can perform high predictability. At the same time, a shared non-locking partition is reserved for the soft real-time tasks. In this way we can target performance improvements based on the data that are frequently used by many tasks in the system. Our experiment results show that the miss rate can be reduced by about 10%~18% compared with that of a statically partitioned cache and by about 24%~40% compared with a dynamic cache using LRU replacement policy.

Key-Words: - Data sharing, Cache partitioning, Embedded database, Real-time

1 Introduction

With the increase of the capacity of the main memory, the embedded databases can reside on faster memory, which are called memory-resident databases. But the gap between the processor and the main memory has made it necessary to resort to the on-chip memory in embedded systems [1]. On-chip SRAM is used as caches for this purpose. However, caches introduce unpredictability in general real-time systems. Thus how to effectively use the caches is crucial to high performance of the systems. Unfortunately, relatively few works on building cache-friendly approaches has been done embedded real-time database systems [3].

There are two key issues when exploiting caches utilization for all tasks for an embedded database in a preemptive, multi-programming environment. The first one is to exploit data locality (cache behavior) for the tasks and eliminate the interferences among them. In fact, it is known that 90% of overall stalls are introduced by the data cache misses [2]. Another key issue is the unpredictable cache behavior. This is a tamper to real-time. The existing approaches on caches for real-time systems are partitioning [6, 9, 10, 12, 13, 15] and/or locking [4, 5, 8] cache segments. Some approaches [14, 15] partition using the modified LRU replacement method or column caching. These approaches are used to eliminate the unpredictability for real-time applications by restricting cache conflictions within the partition owned by the task. Their experimental results show that the performance is improved and their approaches are better than a normal cache adopting LRU replacement policy. But in there scheme, the partitioning is fixed and can not be adjusted at run-time.

In general, we present a data centered cache partitioning approach that allows different tasks to share a locking partition defined by the database manager. In the mean time the hard real-time tasks have their own private partitions, thus, they can perform their jobs with high predictability. The soft real-time tasks will share the non-locking partition reserved to quicken their running speed. Traditional partitioning approaches do not consider locking shared data as a main concern, whereas we target performance improvements based on the data that are frequently used by many tasks in the system.

The remainder of the paper is organized as follows. We describe the data centered cache partitioning scheme in Section 2. We describe the experiment setup about the hardware configuration,

the database server architecture and the workload configuration in Section 3. Experimental results are presented in Section 4. Conclusions and ongoing work is the subject of Section 5.

2 Data Centered Cache Partitioning

In this section we will first describe the general architecture of our data centered cache partitioning scheme. Then we will explain how to specify the size of each partition. We will discuss the impact of low associativity to the system at last.

2.1 General Architecture

The memory architecture model is shown in Fig. 1.

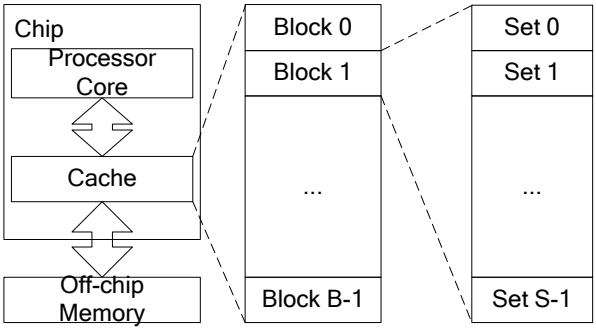

Fig. 1: Memory architecture model

The cache on chip is a W-way set associative one interconnection with the processor core. The size of the cache is S_c and it consists of B blocks whose size are S_b, i.e. $S_c=B*S_b$. The B blocks are further divided into S sets, represented by $S_i(1\le i\le s)$ and every set has W cache blocks ($B=S*W$), a memory address ad is loaded into set: $\left\lfloor \frac{ad}{S_b} \right\rfloor \bmod S$ and it can be put in any of the W cache blocks.

If there is a memory sequence made up by S blocks whose size is S_b and they are all mapped into cache, then it will take up a cache block in every set of S_1, S_2,…S_S. We define the set of these blocks to be a partition set, thus a W-way cache will have W partition sets. Our task is to categorize these partitions and allocate them to the tasks.

The cache space will be partitioned into three parts by the data centered partition algorithm: a shared locking partition, task private partitions and a shared non-locking partition. All these three partitions together form what we call the living condition of the task set. By carefully adjust the sizes of the partitions; we can get an optimal or sub-optimal living condition under a certain configuration of the cache space used by all the tasks. The system sees a task coming or leaving to be a mutation, which is the signal that identifies a turning point for the system to dynamically re-adjust the partition model to accommodate the new task set. The system will dynamically re-adjust the sizes of shared locking partition, task private partition and shared non-locking partition based on the feature of the new task set and come to another optimal or sub-optimal living condition. Figure 2 shows a general cache partition.

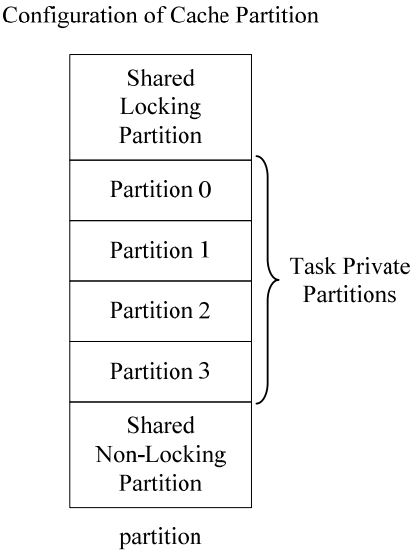

Fig. 2: The Configuration of Cache Partition

2.2 Specifying the Partition Size

The size of shared locking partition is determined by the sharing data in the system, *SHARING*. If a data object is used by two or more tasks in the system, then it is a sharing object, and in an embedded database system, we can get the data objects accessed by the tasks through pre-analysis before the tasks are running and then we can find the sharing number of a particular data object by counting the tasks that access the data object. Suppose there are N data objects in an embedded database system O_1, O_2, O_3, …, O_n, after counting the sharing number of each data object we get N_1, N_2, N_3, …, N_n, the size unit of every data object is partition set size defined earlier, we will take them as S_{ob1}, S_{ob2}, S_{ob3}, …, S_{obn}. Note that S_{obi} means that the size of data object O_i is S_{obi} partition set. We suppose that N_1, N_2, N_3, …, N_n are ordered from the largest to the smallest, if they are not, we can re-arrange the order of the data objects and get such a sorted list.

If N_1=1, then there are no sharing data between the tasks and *SHARING*=0, the size of the shared locking partition is zero; if N_1>1, there are sharing data between the tasks, and if $S_{ob1}>W$, then the size of the data object O_1 is too big for the cache size, so let *SHARING*=W, i.e. allocate all cache spaces to O_1 or part of O_1, we will see from the discussion below that because the hard real-time tasks need their

private partitions and the soft real-time tasks need their shared non-locking partitions, the case *SHARING=W* only temporarily exists to represent the shared locking partition. After the second phase of allocating cache spaces to the tasks, *SHARING=W* will normally disappear from such a system.

If N_1>1 and S_{ob1}>W, then data object O_1 could be put into the cache. We will consider the placement of the second data object O_2, if N_1=1 then data object O_2 is not shared between tasks, so *SHARING*= S_{ob1}, the process is finished. If N_2>1, then we will compare the size of data object O_2, S_{ob2}, with the remaining available cache space W- S_{ob1}, which is like comparing data object O_1 with cache total partition size W, if S_{ob2}>W-S_{ob1}, then S_{ob2} is too much for remaining cache partition, thus, *SHARING=W*, the process is finished, or else S_{ob2}<W-S_{ob1}, then the cache is able to hold data object O_2, so *SHARING*=S_{ob1}+S_{ob2} , we can continue to compare data object O_3 with remaining cache space, repeat the above process until some N_i=1 or *SHARING=W*, the former means that data object O_i is not a shared object, the latter means that the size of shared data object is too much for the cache size.

After computing shared locking partition size SHARING , there are three possible outcomes , as showed in Figure 3.

$$\text{SHARING}=\begin{cases}0, \text{ no shared data objects}\\ S_{ob1}+S_{ob2}+\ldots+S_{obn} \\ \text{all shared objects in cache}\\ W, \text{ shared object size over cache size}\end{cases}$$

Fig. 3: Data Objects Sharing Number.

After computing the shared locking partition size following the process above, we will specify the size for task private partitions based on the task feature. The main factors that feature the tasks are whether it is a hard real-time task or soft real-time task, its deadline D, priority P, its period T, and worst case execution time C. The system chooses different factors to judge the task requirements for the partitions. Here we consider an embedded real-time system in favour of the periodic hard real-time tasks. We assume there are N hard real-time tasks in such a system τ_1, τ_2, τ_3,…, τ_n and their deadlines are D_1, D_2, D_3, …, D_n. The deadlines are ordered from the smallest to the largest. The priority of the tasks is inversely proportional to the deadlines, so it can be represented by the deadlines. We optimistically assume that the data requirements are proportional to the data processing time, i.e. worst case execution time. It means that the worst case execution time C_1, C_2, C_3, …, C_n can be used as an approximation for the data accessed by the task.

Note that the assumption is not applicable to every situation. We only used the method above to describe an approach specifying how to divide the cache spaces into the task private partitions. Another relatively superior approach is to partition the cache based on the utilization of the system $U = \sum_{i=1}^{N} \frac{C_i}{T_i}$, with the goal to minimize the CPU utilization. We can use the dynamic programming algorithm for cache memory partitioning for real-time systems proposed in [20]. The following discussion is based on the assumption we mentioned above.

For the cases in *SHARING=0* and *SHARING*=S_{ob1}+ S_{ob2}+…+ S_{obn}, it is easy to specify the task private partition, the only difference is that the available cache space after the shared partition which is calculated is not the same, one has W partition left, the other has W- S_{ob1}+ S_{ob2}+…+ S_{obn} partition left. We will use C_{left} to represent the available cache space. First we calculate the data cache requirements for every task $\tau_{qi} = w_i \times \frac{C_i}{D_i}$, where τ_{qi} is the data cache requirements, *wi* is weight factor and it is associated with the system, here we assume w_i, C_i is worst case execution time and it can approximately represent the data cache requirements, $\frac{1}{D_i}$ can be viewed as the task priority.

Then we will assign the free partitions left to all the tasks in the system, and the following equation gives the size of task private partition for each task

$$\tau_{ci} = \left\lfloor \frac{\tau_{qi}}{\sum_{i=1}^{n} \tau_{qi}} \times C_{left} \right\rfloor$$

Note that τ_i in this equation takes the lower bound. It is possible that a few partitions will not be assigned to any task. It is free and can be used as shared non-locking partition. And due to the same reason, it is possible that $\tau_{ci} = 0$, it means that we do not reserve a task private partition for real-time task τ_i. If a task τ_i must have its private partition, then we can take back some spaces from the other two partitions and re-assign them to the task using the method below.

For *SHARING=W*, as the shared locking partitions have taken up all the cache spaces, we

have to replace some of them with the task private partitions. It is certain that the least amount of shared data, i.e. the smaller sharing number of the data objects will be replaced first. Here we put forward a straight and simple method. We just use a proportion of 1/2 to split the cache into two equal parts, one for shared locking partition and the other for task private partition. This value could be re-adjusted according to the system workload. For example, suppose that there are lots of data objects whose sharing number is 2 or 3, there will be no significant impact on the system if these data objects are replaced, and we can allocate more cache spaces to task private partition. And if the tasks need little private space of their own and they mainly access the shared data objects, we can allocate more cache spaces to shared locking partition and leave fewer partitions for the tasks of their own.

In order to prevent the cache from thrashing, we will usually reserve a small space for shared non-locking partition to be used by all tasks running in the system. In some embedded processors, like Intel XScale 27x, the processor itself has a minimum cache space to be left for such a purpose. There is 1/16 cache space that can not be locked. So in such a system, we do not have to consider the shared non-locking space, it is fixed and all the other cache spaces can be partitioned into shared locking partitions and task private partitions.

2.3 Low Associativity

This algorithm is the most efficient for a fully associative cache. A cache partition is a cache block in a fully associative cache. For example, a 32KB cache and the cache block size is 32 bytes, then there are 1KB cache blocks, therefore there are 1KB cache partitions, and they can be efficiently partitioned for the task set. For an N-way associative cache, the number of the cache partitions is equal to N. If N is relatively large, say $N \geq 32$, the system can still get benefits from such a partitioning scheme. Although the cache partitions are relatively small compared with a fully associative cache, we can still get a nearly optimal configuration according to the workload of the system. But If N is quite small, like $N \leq 8$, then there are only 8 cache partitions in the system, and every partition is quite large. The algorithm mentioned above will not fulfill a good performance under such a low associative cache.

What we adopted to solve this problem is to carefully arrange the memory layout of the data object so as to let the different parts of the same object lie in the same mapping area in the cache. From [7, 11], we get the inspiration for this approach. The database is responsible for the assignment of the data object in the memory and it is a run-time action, whereas the software based cache partitioning needs additional compiler support and it is a compile-time action.

As the associativity is very low or a directly mapped cache is used, the address will compete for the same block in the cache and it will result in conflict cache misses. If all the addresses of a data object could be mapped into the same cache area, then this cache space will represent a cache partition set in a virtual way. The database will decompose the data object into several parts and each part will reside in the same place in the memory page. Here a memory page is a continuous memory region that maps fully into the cache, i.e. it is equal to the cache size. Figure 4 shows such a direct-mapped cache that is used this way. If a data object is placed in the first 1KB area in every page, then the entire data object will be mapped to the first 1KB space in the cache.

This approach does solve the problem of partitioning the direct-mapped cache, but it goes against the initial purpose of our data centered cache partitioning scheme. Our approach is meant to partition the cache by the number of the ways it has as each way in the set associative cache can be viewed as a direct-mapped cache and every way can be mapped to all the memory addresses. In a direct-mapped cache, however, only a certain area in the memory will be mapped to the cache, so the data objects have to be split to accommodate such cases. So normally we do not use the data centered cache partitioning approach in a low associative cache.

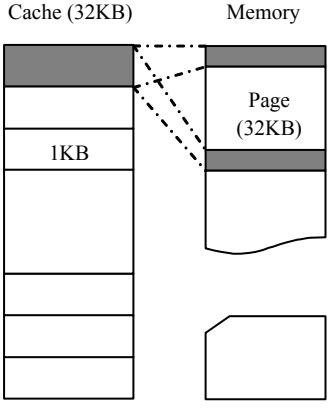

Fig. 4: Data object split in a direct-mapped cache.

3 Experimental Setup

In this section, we describe the setup for our experiments: the simulated hardware platform we run on, the simple real database system we built, and the workload of tasks.

3.1 Hardware Platform

We simulated the cache hardware using Verilog as currently we cannot find a real hardware platform which supports both locking and partitioning. The cache can be locked by lines and it also can be partitioned by its associativity. The smallest partition unit is defined by the cache blocks in the same position of every set in the cache altogether.

We will conduct experiments for data caches commonly used in real-time systems. We will choose 32B-line data caches, whose sizes vary from 16KB to 64KB. For each cache, we will consider a set associative cache from 16-way to 64-way. These are the typical configurations in a modern embedded processor. For example, The Intel XScale 27x embedded processor has a data cache which is 32KB and 32-way associative.

We do not use a low associative cache in our experiment. There is a reason for that. The basic data centered cache partitioning scheme depends on highly associative caches to provide sufficiently small mapping granularities. Highly associative caches, however, are becoming more common, especially in embedded processors. Such associativity is generally implemented with Content Addressable Memories (CAMs) that require an area of the size anywhere from twice to four times larger than the current one but consume less power than traditional associative structures.

3.2 Task Workload

The task workload here consists of two kinds of tasks mentioned above: the periodic tasks and non-periodic tasks.

In our experiments, the number of the periodic tasks is 8. They have their priority P, deadline D, worst case execution time estimated C and period T respectively. We can get the task private partition size for each periodic task based on these parameters. And also the data objects they are accessing are derived. The shared data objects used by all tasks or some of the tasks are recorded.

The embedded system will synthesize a non-periodic task sequence randomly at a rate of 10 tasks per second. Every task generated will first go through the task manager, and meanwhile the proper access pattern information will be acquired by the manager. The shared locking partition will be adjusted if it is needed. If the shared locking partition is enlarged then the shared non-locking partition will be shrank into the same size and vice versa. Each simulation runs for about 10 seconds, so every time there will be about 100 tasks generated.

4 Experimental Results

We now present results from our studies. We first discuss the impact of the size of the cache under certain associativity and compare our scheme with a non-partitioned cache using standard LRU replacement policy and a statically partitioned cache. Then, we show the influence of the associativity of the cache under a certain size and also compare the three schemes.

4.1 Cache Size

The simulations compare the miss rate of the standard LRU replacement policy, the miss rate of the statically partitioning scheme and the miss rate of our partitioning scheme.

Figure 5 illustrates the improvements of our partitioning scheme over the standard LRU replacement policy and the static partitioning scheme. The results are shown in various cache sizes, which range from 16KB to 64 KB. The associativity of the cache is 32-way in all cases.

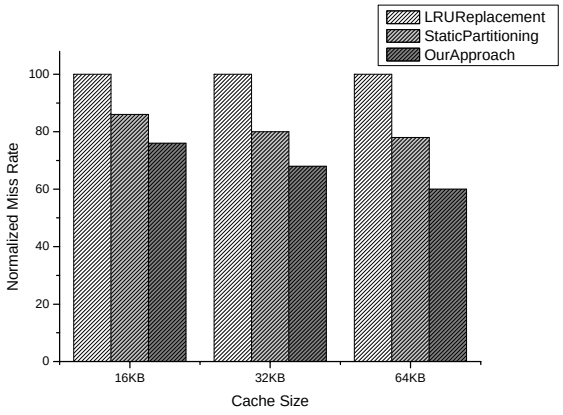

Fig. 5: Miss rate comparison under different cache size with 32-way associativity.

The simulation results shown in Figure 5 demonstrate that the data centered cache partitioning

scheme can further decrease the average miss rate significantly; for a 16KB data cache, the data centered cache partitioning scheme reduces the miss rate to 76% of that of the LRU policy and it is also superior to the static partitioning scheme as the miss rate is reduced by further 10%. The lower miss rate will result in lower CPU utilization, higher predictability and better performance. The system can hold more periodic hard real-time tasks while other soft real-time tasks will receive quicker responses. The main reason for the reduction of the miss rate is that the last two methods eliminate the inter-task interference between tasks in such a preemptive, multitasking environment.

When the cache becomes larger, from 16KB to 32KB, the miss rate will be further reduced. The static partitioning scheme takes another 6% decrease and our approach 8%. As the cache becomes larger, the sizes of the partitions for the hard real-time tasks increase accordingly, thus, the intra-task interference will decrease significantly for each task.

However, when the size of the cache changes into 64KB, the miss rate reduction of static partitioning scheme almost remains the same as in the second case, whereas our scheme can further reduce the miss rate by 8%. It is because this time the cache contains more shared data objects for all running tasks and we do not have to reload these objects into the cache when a task switch happens.

4.2 Cache Associativity

Now we focus on the cache associativity impact on the miss rate. We also compare the miss rate of the standard LRU replacement policy, the miss rate of the statically partitioning scheme and the miss rate of our partitioning scheme.

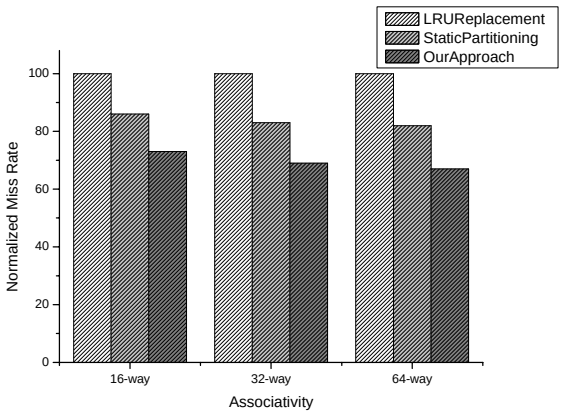

Fig. 6: Miss rate comparison under different cache associativity with 32KB cache size.

Figure 6 shows the results of the simulation running with our partitioning scheme, the standard LRU replacement policy, and the static partitioning scheme. The results are shown for various cache associativities, which range from 16-way to 64-way. The size of the cache remains 32KB in all cases.

First it can be seen that our partitioning scheme can further decrease the average miss rate significantly; for a 16-way data cache, the data centered cache partitioning scheme reduces the miss rate to 73% of that of the LRU policy and it also has better performance over the static partitioning scheme as the miss rate is reduced by further 13%. It shares the same reason as the one we discussed above. The inter-task interference is eliminated between tasks in such a preemptive, multitasking environment.

We can also see that when the associativity is increased, the miss rate drops further down. It is mainly because in a more highly associative cache the configuration can be more flexible as there are more basic units to be allocated to each partition. However, when the associativity is increased, the speed of the reduction of the miss rate slows down. For example, from 16-way to 32-way, the miss rate is reduced by 4%, but from 32-way to 64-way, the miss rate is only reduced by 2%. The main reason is that we only have 8 periodic hard real-time tasks in the system, so a 16-way 32KB cache can satisfy this set of workload pretty well. Using a more highly associative cache does not help much in such a situation.

5 Conclusion

In this paper, we present a data centered cache partitioning approach that allows different tasks to share a shared locking partition defined by the database manager. In the meantime it allows hard real-time tasks to have a private partition of their own, thus, hard real-time tasks can perform their jobs with high predictability. It also reserves a shared non-locking partition for the soft real-time tasks to accelerate their running speed. Our experiment result shows that the miss rate can be reduced by about 10%~18% compared with the miss rate of a statically partitioned cache and by about 24%~40% compared with a dynamic cache using LRU replacement policy if the data centered partitioning scheme is used in a real-time database system. The lower miss rate will result in lower

CPU utilization, higher predictability and better performance.

For our future work, we plan to improve our approach in several aspects. First, the cache is partitioned at the granularity of columns, which may lower the utilization of the cache. We plan to support partitioning the cache at a lower level of granularity in order to solve this problem. Second, we will combine our approach with other partitioning schemes to achieve a better partition between the database task and other tasks. Last, we need to analyze the performance of our approach when the associativity of the cache is very low.

References:

[1] Wind River Inc, *High availability design for embedded systems*, Technical report, http://www.windriver.com/- whitepapers/high availability design.html.

[2] J. Pisharath et al, Data windows: a data-centric approach for query execution in memory-resident databases, *In Proc of the Design, Automation and Test in Europe Conference and Exhibition 2004 (DATE'04)*, 2004, pp. 21352-21353, IEEE Computer Society.

[3] P. Trancoso and J. Torrellas, Cache Optimization for Memory-Resident Decision Support Commercial Workloads, *In Proc of the International Conference on Computer Design1999 (ICCD'99)*, 1999, pp. 546-554, IEEE Computer Society.

[4] I. Puaut and D. Decotigny, Low-complexity algorithms for static cache locking in multitasking hard real-time systems, *In Proc of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, 2002, pp. 114-123.

[5] M. Campoy et al, Static use of locking caches in multitask premptive real-time systems, I*n Proc of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, 2002, pp. 114-123.

[6] X. Vera et al, Data caches in multitasking hard real-time systems, *In Proc of the 24th IEEE Real-Time Systems Symposium (RTSS03)*, 2003, pp. 154-165.

[7] F.Mueller, Compiler support for software-based cache partitioning, *In Proc of ACM Workshop on Languages, Compilers and Tools for Real-Time Systems (LCTES'95)*, 1995, pp. 125-133, ACM Press.

[8] X. Vera et al, Data cache locking for higher program predictability, *In Proc of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, 2003, pp. 272-282.

[9] G. E. Suh et al, A new memory monitoring scheme for memory-aware scheduling and partitioning, *In Proc of the 8th International Symposium on High-Performance Computer Architecture*, 2002, pp. 117-128.

[10] D. B. Kirk, Smart (strategic memory allocation for real-time) cache design, *In Proc of the 10th IEEE Real-Time Systems Symposium (RTSS'89)*, 1989, pp. 229-237.

[11] J. Liedtke et al, OS-controlled cache predictability for real-time systems, *In Proc of Real-Time Technology and Applications Symposium*, 1997, pp. 213-224.

[12] G. E. Suh et al, Dynamic Partitioning of Shared Cache Memory, *The Journal of Supercomputing*, Vol.28, No.1, 2004, pp. 7-26.

[13] J. E. Sasinowski et al, A Dynamic Programming Algorithm for Cache Memory Partitioning for Real-Time Systems, *IEEE Transactions on Computers*, Vol.42, No.8, 1993, pp. 997-1001.

[14] L. Rudolph et al, Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches, *In Proc of 37th Conference on Design Automation (DAC'00)*, 2000, pp. 416-420.

[15] D. Chiou et al, Dynamic Cache Partitioning via Columnization, Proceedings of Design Automation Conference, 2000.